Collections et itérateurs

M. Schinz, J. Sam, J.-C. Chappelier

version 1.0

Résumé

Le but de ce document est de brièvement présenter la notion d'*itérateur* en Java, notion importante dans ce projet pour pouvoir supprimer des éléments d'une collection tout en la parcourant.

1 Collections

En programmation, il est souvent nécessaire de manipuler un ensemble de données comme un tout. Nous en avons vu des exemples dans nos MOOCs précédents au travers de la notion de *tableau* ou de celle de *liste*. On appelle « *collection* » un objet servant de conteneur à d'autres objets. La bibliothèque standard Java fournit un certain nombre de collections dans ce qui s'appelle le « *Java Collections Framework* » (JCF ; paquetage `java.util`), telles que :

- les tableaux ;
- les listes et leurs variantes : piles, files, « *double ended queues* » ;
- les ensembles ;
- les tables associatives.

Pour chaque type de collection (liste, ensemble, etc.), l'API Java contient généralement :

- une interface, qui décrit les opérations offertes par la collection en question ;
- plusieurs classes implémentant l'interface et qui sont les mises en œuvre de la collection, ayant chacune leurs caractéristiques propres.
  Par exemple, une liste peut être mise en œuvre au moyen d'un tableau dans lequel les éléments sont stockés côte à côte (`ArrayList`), ou au moyen de nœuds chaînés entre eux via des références (`LinkedList`).

Ces classes et intefaces sont paramétrées par le type des données qu'elles contiennent, par exemple : `List<Integer>` pour une liste d'entiers, `List<String>` pour une liste de chaînes de caractères etc.

De plus, il arrive parfois que les classes de mise en œuvre, ou en tout cas certaines d'entre elles, héritent d'une classe abstraite fournissant le code commun.

La figure 1 présente une vision simplifiée de la hiérarchie pour la notion de liste.

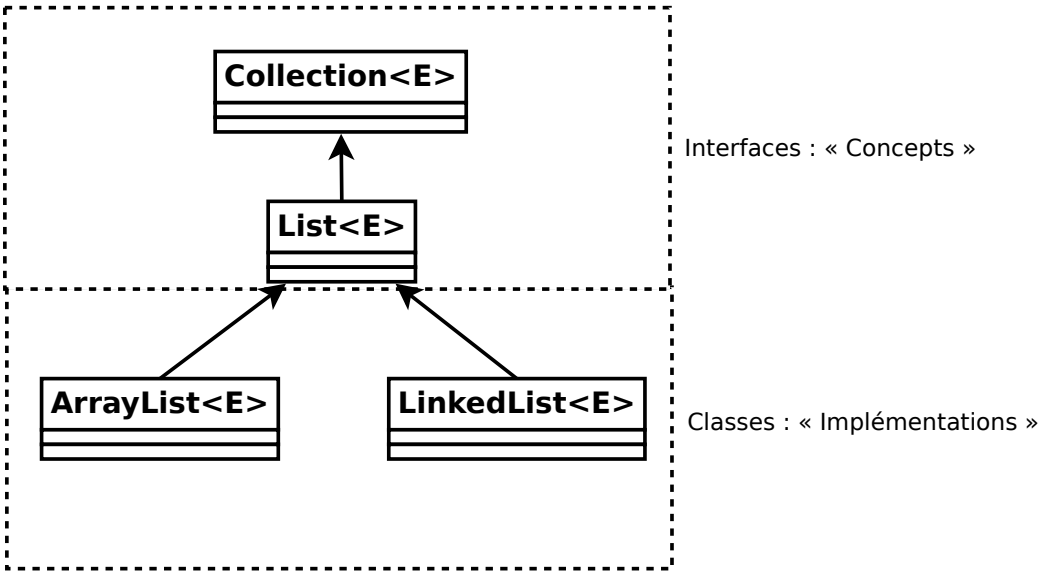

FIGURE 1 – Hiérarchie simplifiée pour la notion de «liste».

En plus des méthodes définies dans les interfaces des différentes collections, on trouve des méthodes statiques relatives aux collections dans les classes `Collections` et `Arrays`. Ces deux classes ont pour seul but de regrouper des méthodes statiques, et ne sont donc pas instanciables (leur constructeur est privé).

Attention : ne pas confondre l'*interface* `Collection` (sans `s`) et la *classe* `Collections` (avec un `s`) !

Chaque type de collection a ses caractéristiques propres, ses avantages et ses inconvénients. Par exemple, une `ArrayList` offre de meilleures performances qu'une `LinkedList` pour accéder directement à un élément, mais est moins efficace pour la suppression d'un élément quelconque. Le choix de la bonne collection à utiliser dans un cas particulier dépend donc de ce que l'on veut en faire.

Les références citées à la fin de ce document vous permettront d'approfondir sur le sujet si vous le souhaitez. La suite de ce document ne se focalise que sur ce qu'il vous est nécessaire de savoir pour ce projet.

2 Parcours d'une collection

Il est très fréquent d'avoir à parcourir les éléments d'une collection. Par exemple, admettons que l'on désire parcourir une liste de chaînes de caractères pour afficher ses éléments à l'écran. Une première — mais très mauvaise — idée consiste à utiliser une boucle `for` et la méthode `get()`, comme pour un tableau :

```
List<String> l = new LinkedList<String>(
    Arrays.asList ("bonjour", "à", "tous")
);
```

```
for (int i = 0; i < l.size(); ++i)
    System.out.println(l.get(i));
```

Cette solution est mauvaise car, dans le cas des listes chaînées (`LinkedList`), la méthode `get()` peut nécessiter de reparcourir tous les `n` éléments de la liste. La boucle d'impression peut donc dans certaines situations défavorables nécessiter pour chaque élément de la liste, de reparcourir à nouveau toute la liste. Ceci est clairement insatisfaisant sachant que chaque élément n'a en réalité besoin d'être examiné qu'une seule fois !

Pour faire mieux, on peut utiliser la boucle `for-each`, car les listes — et d'autres collections — peuvent être parcourues ainsi de façon efficace. Notre boucle peut donc se récrire comme suit :

```
List<String> l = new LinkedList<String>(
    Arrays.asList ("bonjour", "à", "tous")
);
for (String s : l)
    System.out.println(s);
```

Dans le cas des listes en tout cas, cette boucle ne va parcourir qu'une seule fois les éléments de la liste, même avec les listes chaînées.
Comment est-ce possible ?
Grâce à la notion d'itérateur !

2.1 Itérateurs

Un itérateur (*iterator* ou plus rarement curseur, *cursor*) est un objet qui désigne un élément d'une collection. Un itérateur permet d'une part d'obtenir l'élément qu'il désigne, et sait d'autre part se déplacer efficacement sur l'élément suivant — et parfois précédent — de la collection.

Dans la bibliothèque Java, le concept d'itérateur est décrit par l'interface générique `Iterator` :

```
public interface Iterator<E>
```

Son paramètre de type `E` représente le type des éléments de la collection parcourue par l'itérateur.

L'interface `Iterator` est très simple et ne contient que trois méthodes :

- `boolean hasNext()` qui retourne vrai si et seulement si il reste au moins un élément dans la collection après celui désigné par l'itérateur ;
- `E next()` qui avance l'itérateur sur l'élément suivant et retourne cet élément, ou lance une exception s'il n'y a pas d'élément suivant ;
- et `void remove()` qui supprime le dernier élément retourné par `next()`, ou lance une exception si `next()` n'a pas encore été appelée ou si `remove()` a déjà été appelée une fois depuis le dernier appel à `next()`.

Les collections dont on peut parcourir les éléments offrent toutes une méthode `iterator()` permettant d'obtenir un nouvel itérateur à partir duquel on peut accéder au premier élément (en invoquant la méthode `next()`). Au moyen de cette méthode, notre boucle précédente peut s'écrire également ainsi :

```
List<String> l = new LinkedList<String>(
    Arrays.asList ("bonjour", "à", "tous")
);
Iterator<String> i = l.iterator();
while (i.hasNext()) {
    String c = i.next();
    System.out.println(c);
}
```

Nous connaissons maintenant deux techniques (efficaces) pour parcourir une liste : la boucle `for-each` et les itérateurs. Laquelle préférer ?

En termes d'efficacité, ces deux techniques sont rigoureusement équivalentes, la boucle `for-each` étant récrite par le compilateur Java en une boucle basée sur un itérateur. La boucle `for-each` est, par contre, plus concise et facile à comprendre ; on préfèrera donc l'écrire de cette façon plutôt que d'expliciter les itérateurs. Cependant, la boucle `for-each` est moins générale (nous en avons d'ailleurs étudié un certain nombre de limitations dans nos MOOCs précédents). Une de ses limitations importantes est qu'il n'est pas possible de supprimer un élément de la collection lors du parcours. Par exemple, le code suivant :

```
List<String> l = new LinkedList<String>(
    Arrays.asList ("bonjour", "hello", "à", "tous")
);
for (String c : l) {
    if (c.equals("hello"))
        l.remove(c);
}
```

va lancer une `java.util.ConcurrentModificationException`. Une façon simple de contourner ce problème est de recourir à la notion d'itérateur. Le code devient alors :

```
List<String> l = new LinkedList<String>(
    Arrays.asList ("bonjour", "hello", "à", "tous")
);
Iterator<String> i = l.iterator();
while (i.hasNext()) {
    String c = i.next();
    if (c.equals("hello")) i.remove();
}
```

C'est donc comme cela qu'il vous faudra, le moment venu, supprimer des éléments dans les collections du projet.

Références

Ce document est fortement inspiré du cours EPFL « *Pratique de la programmation orientée-objet (CS108)* » du Dr. M. Schinz.

Différents documents fournis par Oracle au sujet des collections, en particulier :

- le tutoriel *Java Collections Framework* [Lien] ;
- *Collections Framework Overview* [Lien] ;
- *Outline of the Collections Framework* [Lien].

La documentation de l'API Java, en particulier les classes et interfaces suivantes :

- les interfaces `java.util.Collection` et `java.util.List` ;
- les classes `java.util.Collections` et `java.util.Arrays` ;
- les classes `java.util.ArrayList` et `java.util.LinkedList` ;
- l'interface `java.util.Iterator`.